A Generic Rendering System

Jürgen Döllner and Klaus Hinrichs

Abstract – We describe the software architecture of a rendering system that follows a pragmatic approach to integrating and bundling the power of different low-level rendering systems within an object-oriented framework. The generic rendering system provides higher-level abstractions to existing rendering systems and serves as a framework for developing new rendering techniques. It wraps the functionality of several, widely used rendering systems, defines a unified, object-oriented application programming interface, and provides an extensible, customizable apparatus for evaluating and interpreting hierarchical scene information. As a fundamental property, individual features of a specific rendering system can be integrated into the generic rendering system in a transparent way. The system is based on a state machine, called *engine*, which operates on *rendering components*. Four major categories of rendering components constitute the generic rendering system: *shapes* represent geometries; *attributes* specify properties assigned to geometries and scenes; *handlers* encapsulate rendering algorithms, and *techniques* represent evaluation strategies for rendering components. As a proof of concept, we have implemented the described software architecture by the *Virtual Rendering System* which currently wraps OpenGL, Radiance, POV Ray, and RenderMan.

Index Terms – Rendering systems, object-oriented graphics, generic rendering, rendering framework, multi-pass rendering.

1 INTRODUCTION

Rendering systems represent the basis for computer graphics applications in such diverse fields as CAD, CAE, medical and scientific visualization, and entertainment. Rendering systems differ with respect to the type of illumination model (e.g., local illumination models vs. global illumination models), support for geometric modeling (e.g., polygon-based modeling, implicit surfaces, volume-based modeling), and support for animation and interaction. In the last 15 years, many rendering systems emerged, but only few of them have been matured and established themselves as industry standards.

Rendering systems result from long and complex analysis, design, and implementation processes, and thus embody a huge amount of both practical and theoretical work: The implementation of a rendering system requires efficient algorithms and data structures for rendering, modeling, optimization, and animation; a deep knowledge of the underlying principles of illumination and geometry; and expert knowledge about specialized graphics hardware. In addition, designing the software architecture of a rendering system is becoming more complex due to manifold developments of new real-time, photorealistic, and non-photorealistic rendering techniques. It would be not feasible to write one new, object-oriented rendering system from scratch that supports the whole bandwidth of rendering techniques and graphics hardware covered by existing rendering systems. This has been the motivation for us to design the *generic rendering system* which wraps the functionality of existing rendering systems in an object-oriented way, defines a generic, object-oriented interface that does not suppress but preserve their individual features, and provides an extensible and easy to customize apparatus for constructing and evaluating hierarchical scene descriptions.

The generic rendering system defines a generic and extensible class model for rendering components, based on a generalized rendering pipeline that defines rendering as a recursive process of in-

terpretation and evaluation of rendering components. Rendering components encompass *shapes* which represent 2D and 3D geometries, *attributes* which describe the visual and geometric properties such as the appearance of shapes and scenes, *handlers* which represent rendering algorithms, and *techniques* which represent strategies for interpreting and evaluating rendering components. *Engines* manage the evaluation process; they use techniques to select appropriate handlers and delegate the tasks of interpretation and evaluation to them.

The whole system is more than its parts: the generic rendering system can also extend rendering and modeling techniques originally not available in a low-level rendering system. The built-in functionality helps to reduce the implementation effort, and the handler and technique design patterns make it possible to abstract and encapsulate complex rendering algorithms and data structures, for example, shadow algorithms, real-time lighting and shading techniques, or image-based CSG modeling. To integrate unique features of a low-level rendering system, the core set of rendering components can be extended; there is no preference for built-in rendering components. Applications can use the generic rendering system, for example, for both high-quality rendering and real-time rendering within one single framework and without having to modify the source code or loosing unique features of the concrete rendering systems. Consequently, the generic rendering system improves the usability and extensibility of today's low-level rendering systems.

To prove the feasibility of the described software architecture, we have developed the *Virtual Rendering System*. VRS, a portable C++ toolkit, is currently wrapping OpenGL [42], the lighting simulation and rendering system Radiance [40], POV-Ray [26], and Pixar's RenderMan [38]. In particular, the extensive adaptation for OpenGL provides a higher level of abstraction for complex OpenGL programming techniques such as multitexturing, bump mapping, and shadow maps. There is no significant performance penalty compared to programs directly using OpenGL.

Figure 1 shows two snapshots produced by VRS. They are taken from a movie, which explains, visualizes and animates a complex polyhedron, one of the so-called Coxeter Polyhedra; the polyhedron has hidden symmetries, which become visible when rotating and projecting the polyhedron's edges onto a plane. The application uses the generic rendering system to model and render the scene based on a single scene graph. The OpenGL engine is used for designing the animation, whereas the POV-Ray engine is used to produce the final video sequence.

The remainder of this paper is structured as follows: Section 2 discusses related work; Section 3 introduces the software architecture of the generic rendering system; Section 4 defines the rendering components of our approach; Section 5 and Section 6 explains how rendering components are evaluated and hierarchically modeled; Section 7 gives details about integrating new shape types; Section 8 discusses details of the design of attributes; Section 9 extends our approach towards multi-pass rendering; Section 10 illustrates applications of the generic systems; and Section 11 gives conclusions.

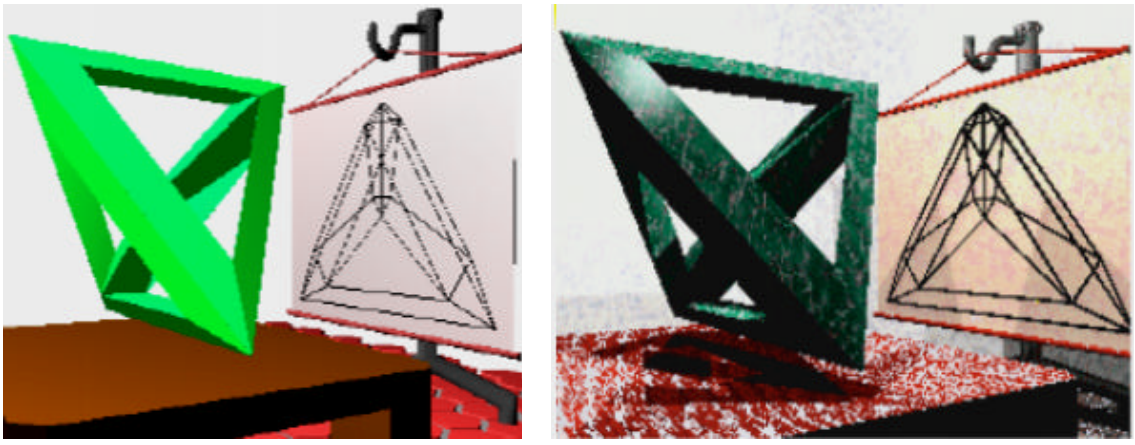


Figure 1. A scene modeled with the generic rendering system, rendered with the OpenGL engine (left) and with the POV-Ray engine (right). The images have been produced based on the same source code by exchanging the rendering engines; the scene graph contains attributes specific to OpenGL and POV-Ray.

2 RENDERING SYSTEMS – AN OVERVIEW

This section discusses related work. We briefly summarize the characteristics of standard rendering systems and analyze the major limitations of their software architecture.

2.1 Rendering Systems

White-box rendering systems document their internal design and provide access to their implementation. In general, they are delivered as libraries or frameworks, which can be extended and redesigned. *Black-box rendering systems*, in contrast, hide their software architecture and can be used exclusively through a well-defined programming interface such as a scene description language. *Gray-box rendering systems* do not give full access to the system's design and implementation. In general, developers can extend at least some parts of a gray-box system, for example, by implementing derived classes. A detailed reference for object-oriented and component-oriented software engineering can be found in Szyperski [36].

Vision [32] provides an extensible white-box framework for implementing rendering techniques focusing on global illumination calculations. The Vision architecture completely separates geometry objects and their attribute objects using object-oriented design even at a low level in the system architecture [31]. In our work, the separation between geometric primitives and attributes has been extended: The generic rendering system provides a generic attribute management for handling rendering-system dependent and rendering-technique dependent attribute types. In addition, we separate rendering algorithms from shape classes and attribute classes. Furthermore, we provide a generic concept for evaluating scene graphs.

GRAMS [11], an object-oriented white-box rendering framework, appears to be the first system explicitly supporting different rendering techniques (e.g., real-time rendering, ray-tracing). It distinguishes between a rendering layer and a graphics layer. This separation ensures that new functionality can be added to a layer without altering the other layer. The GRAMS architecture separates rendering algorithms from geometric primitives. Rendering algorithms are selected for a target low-level rendering system using rendering efficiency as criterion [12]. The generic rendering system extends the separation towards algorithms evaluating attributes and multi-pass rendering techniques.

Generic3D [4] defines an extensible, object-oriented white-box rendering library. It consists of a collection of classes which can be combined and subclassed to implement application-specific rendering systems, so called customized graphics kernels. Generic uses OpenGL for real-time rendering, but does not support any other third-party rendering system; it is intended as a framework for implementing new rendering systems.

The BOOGA project [1] develops a white-box component-based software architecture for graphics applications. The system defines three layers, a basis layer, a framework layer, and a component layer. The system provides a high degree of extensibility because each layer can be extended independently using inheritance or template instantiation. For example, the evaluation strategy for scene graphs is based on the visitor design pattern and implemented by so called renderer components.

OpenInventor [33] represents a sophisticated object-oriented rendering library for interactive 3D graphics designed as a gray-box system. It has introduced the classical concept of a scene graph, which has been adopted by many other systems (e.g., Java3D). As a common characteristic, order and arrangement of rendering primitives in the scene graph reflect the order in which rendering primitives are sent through the rendering pipeline. OpenInventor concentrates on real-time rendering and does not support other low-level rendering systems or rendering techniques (e.g., photorealistic and non-photorealistic rendering).

Java 3D [35], a gray-box rendering library, defines classes for graphical attributes and geometric objects focusing on real-time computer graphics. Java 3D's high-level constructs (e.g., scene graph, view model based on physical body and physical environment, geometry compression, spatial sound etc.) are designed for constructing virtual worlds and are well suited for interactive 3D graphics. Java 3D defines a core set of shapes but does not permit one to add new types of shapes unless they are

reduced to elementary shapes of Java3D, i.e., there is no access to the capabilities of the underlying low-level 3D rendering library. This, however, restricts the extensibility, in particular if application-specific shape types are supported by the 3D hardware or the low-level 3D rendering system. New rendering techniques cannot be integrated into Java 3D because the evaluation process applied to a scene graph cannot be redefined or specialized. Java 3D claims to be a "fourth-generation" 3D API [23] and to synthesize its low-level graphics constructs from the best ideas found in low-level APIs such as Direct3D, OpenGL, QuickDraw3D, and XGL. The implied restriction is that sophisticated 3D graphics applications requiring advanced features of, say OpenGL, can hardly be implemented because there is no access to specialized but important rendering features, e.g., OpenGL P-buffer rendering, per-fragment operations, and multi-texturing.

Many other white-box rendering systems apply object-oriented software design principles. Examples include the MRT [14] toolkit or TBAG [13], a functional approach to interactive and animated graphics programming, and GROOP [6], a system concentrating on simplicity using an actor-stage metaphor in the object model.

There is a large number of black-box rendering systems, for which less is known about their software architecture, for example, the Blue Moon Rendering Tools [15] implementing the Pixar RenderMan [38] standard or POV-Ray, a popular ray-tracing system. RenderMan established a well-defined, powerful scene description specification, concentrating on an abstract specification of scene objects and their attributes; RenderMan makes no assumption about the concrete rendering technique used to render a scene. The RenderMan interface has influenced strongly our shape and attribute design.

2.2 Limitations of the Software Architecture of Rendering Systems

As a common characteristic, the aforementioned white-box and gray-box rendering systems provide efficient object models aligned towards the *implementation* of their underlying rendering paradigm. Limitations resulting from this include:

Restricted Portability. Graphics applications based on one rendering system cannot be adapted easily to another rendering system because the source code must be redesigned completely. For example, an application based on OpenInventor cannot be transformed to RenderMan's RIB scene description language. Consequently the application developer will less likely experiment with different rendering systems and techniques.

Restricted Extensibility. The software architecture of most rendering systems assumes a specific kind of rendering technique. In general, the integration of new rendering techniques, for example non-photorealistic rendering techniques, can be achieved neither technically nor economically because too many aspects of the system architecture would have to be redesigned. Most rendering systems do not support extensibility by integrating external libraries. This is only possible if the rendering system has an open architecture and provides a plug-in concept. Otherwise, extensibility is restricted and external rendering libraries cannot be reused.

Complex application programming interface. The application programming interfaces of low-level rendering systems consist of a multitude of low-level data structures and commands (e.g. OpenGL [42], Direct3D [22]) which developers have to realize, to read and to understand. Thus, implementing applications on top of existing rendering systems requires deep and detailed knowledge of the specific interface of the system.

The object model of the generic rendering system is not aligned to a specific rendering paradigm. Instead, the generic rendering system identifies similarities in the object models of gray-box and white-box rendering systems and presents a uniform and generic software architecture. It is designed as a white-box rendering system that will most likely be used as gray-box system.

To support portability, the generic rendering system encapsulates different, autonomous low-level rendering systems under a uniform application programming interface. To support extensibility, the generic rendering system allows developers to integrate new rendering techniques and rendering

libraries in a straightforward way. Thus, developers can take advantage of many specialized, sophisticated rendering libraries, e.g., collision detection libraries, OpenGL related extensions such as OpenGL Optimizer [30] and the tubing and extrusion library GLE [39], or geometric modeling software such as blob trees [37]. To support ease of use, the generic rendering system concentrates on an object-oriented and declarative application programming interface. Object-oriented software architectures have proved themselves useful for higher-level rendering systems [6] and form a prerequisite for component-based software architectures. Declarative interfaces are generally easier to understand and less bound to a specific implementation. As a side effect, the generic rendering system simplifies the understanding and usage of the integrated low-level rendering systems.

3 SOFTWARE ARCHITECTURE OF THE GENERIC RENDERING SYSTEM

In this section, we briefly define key terminology and introduce the software architecture of the generic rendering system.

3.1 Terminology

The *software architecture* of a software system describes that system in terms of software components, their composition, and the interactions among those components [29]. The architecture is specified by models consisting of model elements that describe static, dynamic, and physical aspects of the software system. In general, models defined by the Unified Modeling Language UML [27] are deployed to specify the architecture of object-oriented software systems; we will use the UML notation throughout the paper.

By *rendering system* we understand software and hardware that synthesize images based on the geometric descriptions of real or imaginary objects. A rendering system can be implemented as rendering library or rendering framework. A *rendering library* provides a collection of general-purpose classes and functions used to develop potentially any kind of graphics application. Examples include OpenInventor, Java3D, and OpenGL. A *rendering framework* consists of a collection of classes and functions that cooperate in order to implement a certain kind of application. To build an application, a framework is specialized and extended. Examples include the BOOGA framework, the Vision framework, or the Generic-3D framework.

By *rendering* we understand, in a wider sense, the translation of data from one representation into another representation. In computer graphics, rendering denotes the process of synthesizing images, i.e., the translation of geometry, controlled by associated graphics attributes, to the image medium. More general, rendering includes interpretation and evaluation of rendering components for a target medium. The strategy for interpreting and evaluating rendering components depends on that target medium. This broader definition of rendering conforms to the original meaning of 'to render', to cause to be or become or to translate.

We introduce the term *generic rendering system* to denote a kind of rendering system that generalizes a number of lower-level rendering systems, expressing their commonalties and differences within a framework. “Generic” is used in its original meaning of “relating to, characteristic of a whole group or class” (Webster's New Encyclopedic Dictionary). To implement the generic render-

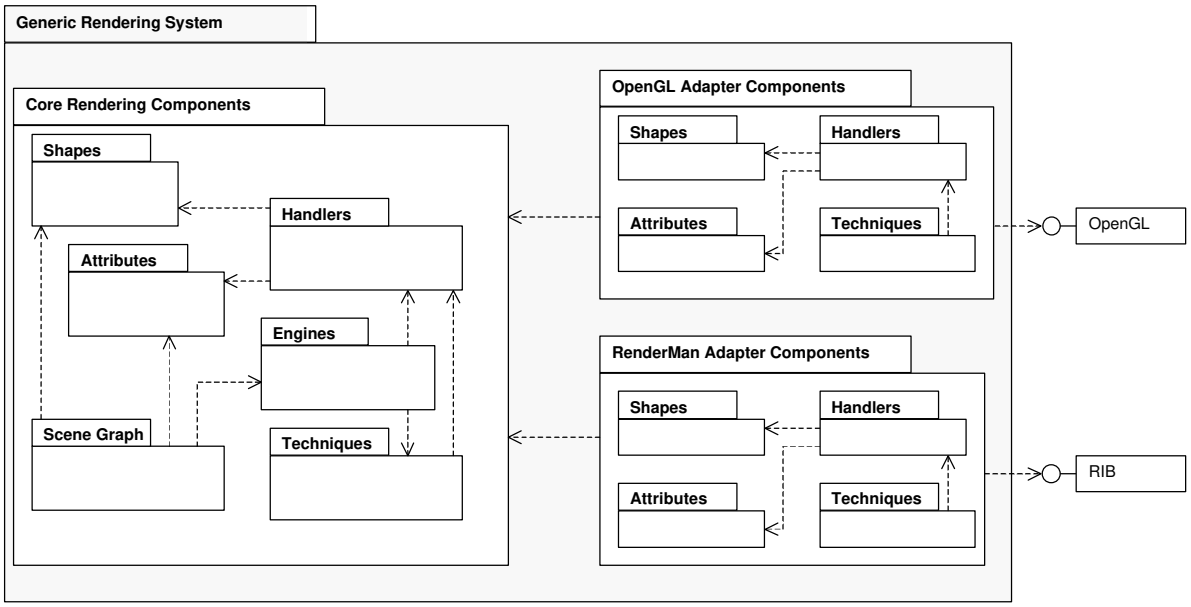


Figure 2. Main software packages of the generic rendering system.

ing system, lower-level rendering systems need to be integrated into the framework that defines the key abstractions and key design patterns to incorporate and access the individual features of a lower-level rendering system.

3.2 Software Architecture

The main packages that constitute the software architecture of the generic rendering system and their dependencies are outlined in Figure 2. The generic rendering system consists of a package of core rendering components common to all low-level rendering systems, and packages that contain specialized rendering components for a given low-level rendering system.

Among the core rendering components, shapes, attributes, and scene graph components represent the components used by application developers. Their implementation is based on handlers (i.e., rendering algorithms), techniques (i.e., rendering strategies), and engines. Shapes, attributes and engines are hierarchically organized and aggregated, composed by a scene graph. Handlers and techniques, however, are typically managed by engines and thus are not visible to the developer. Handlers are responsible for interpreting and evaluating shapes and attributes. Shapes and attributes do not know which handlers will be applied to them. To extend the capabilities and functionality of the generic rendering system, new components may be introduced in all sub packages.

To integrate a low-level rendering system into the generic rendering system, specialized handlers, techniques, and possibly attributes must be defined; they are called *adapter components*. In general, these specialized classes inherit from base classes defined in the core package. Specialized handlers may interpret existing attribute classes and shapes classes, as well as specialized attribute classes and shape classes. Each adapter package uses the appropriate programming interface of the corresponding low-level rendering system, e.g., the OpenGL application programming interface or the RenderMan interface RIB.

4 RENDERING COMPONENTS

This section introduces the rendering components of the generic rendering system. The key design elements are the object-based state machine, represented by the engine, and handlers and techniques, which separate rendering algorithms from shapes and attributes.

The major categories of rendering components of the generic rendering system are classified into a few categories depending on similar behavior, i.e., the categories are based on a logical (and not implementation-driven) decomposition. The categories and their uses-relationships are depicted in Figure 3.

Shapes

Shapes denote objects that are perceived as entities of the target medium. For a visual medium, these objects are geometric objects; in a sound rendering system, shapes would be sounds. Typical shape classes include 2D and 3D geometries such as polygonal meshes, free-form surfaces, curves, and images. Shape classes, however, do not define or reference the rendering algorithms that map them to a low-level rendering system, nor do they define the attributes that control or specify the mapping process.

Attributes

Attributes denote modifiers that control or specify the evaluation of shapes for a target medium. For a visual medium, attributes include all kinds of graphical attributes such as color, texture, or material properties. Geometric transformations are specialized attributes that control the transformation of shapes. Attribute classes, however, do not define or reference the rendering algorithms that map them to a low-level rendering system. The rendering technique used to evaluate a collection of rendering components determines which attributes are actually considered and how they are interpreted.

Handlers

Handlers denote rendering algorithms that interpret shapes and attributes. A handler class is responsible for a specific shape or attribute class, called its *target*, and provides a kind of rendering functionality, called the *service*. A sphere rendering algorithm, for example, has sphere shapes as targets and provides the service 3D-rendering. Handlers are represented as independent objects; they decouple rendering functionality from class descriptions of shapes and attributes. A handler encapsulates an algorithm, i.e., a fragment of code, in an object that can be plugged into and removed from the generic rendering system. Since handlers can be created and associated with engines dynamically, handlers shift also the binding of rendering functionality to shapes and attributes from compile time to run time.

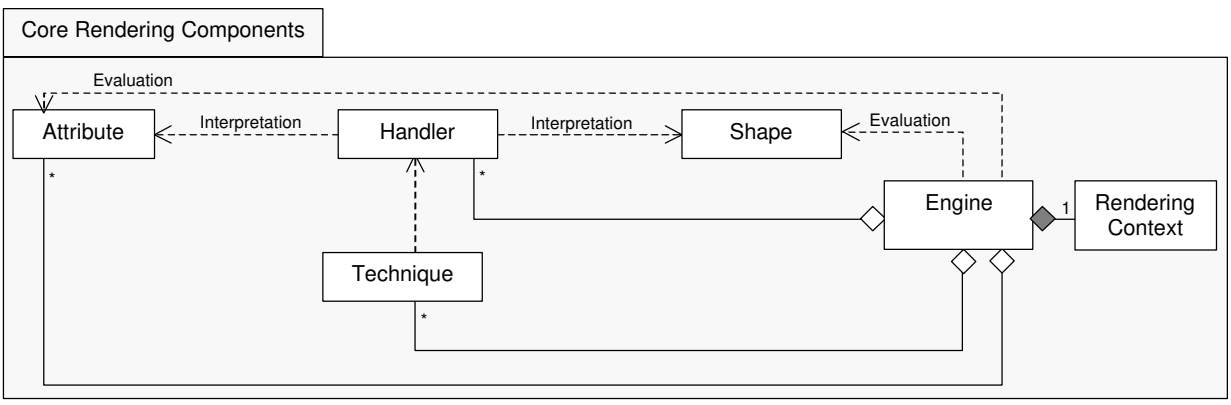


Figure 3. Uses associations and part-of associations between core rendering components.

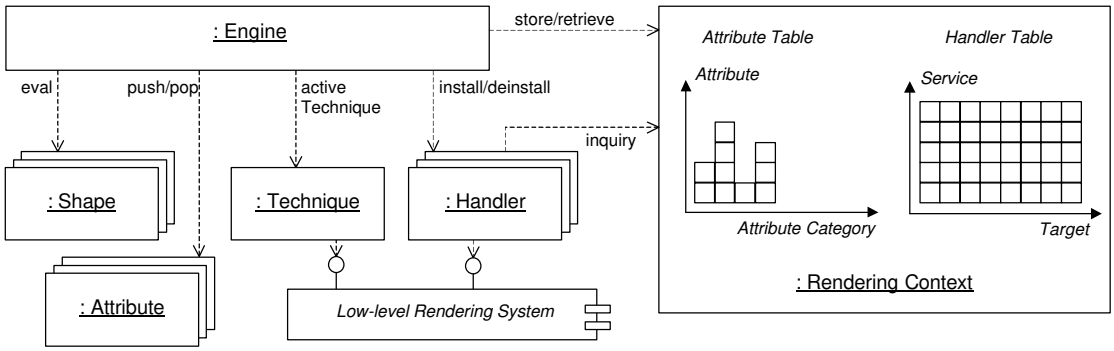


Figure 4. Tasks of an engine to process rendering components.

Techniques

Techniques denote strategies to process a sequence of rendering components. Techniques determine suitable handlers for shapes and attributes sent to an engine, and delegate the execution to these handlers. Techniques define when which services are called. Techniques are represented as independent objects; they enable the generic rendering system to model any kind of evaluation for a sequence of rendering components. A typical evaluation process is the image synthesis process: image synthesis techniques search for painting services and control the image generation.

Engines

Engines manage the rendering context and serve as a compact interface to the generic rendering system. They trigger the evaluation of shapes and manage attributes. Nodes of a scene graph send their *content objects*, i.e., the rendering objects they contain, to an engine; the engine has the role of a visitor traversing through the scene graph. The tasks performed by an engine to process rendering components are illustrated in the object diagram in Figure 4.

5 EVALUATING RENDERING COMPONENTS

The rendering components constitute the atoms of the generic rendering system. This section explains how sequences of rendering components are evaluated; the next Section will explain how sequences of rendering components result from a hierarchical modeling scheme.

5.1 Rendering Context

An engine maintains a collection of active attributes and handlers, stored in an associated *rendering context*. The rendering context consists of an attribute table and a handler table. The engine delegates the evaluation of shapes as well as the activation respectively deactivation of attributes to its active technique, which in turn delegates these tasks to appropriate handlers. Both, handlers and techniques are generally interfaced with a specific low-level rendering system. Engines are represented as independent objects to enable the generic rendering system to model rendering strategies explicitly.

The *attribute table* allocates for each *attribute category* an object container (Figure 5). In general, the attribute category corresponds to the class an attribute object belongs to. For example, a material attribute belongs to the category "material". Specialized attributes can be stored in a single container, if they define a common attribute category; subclasses of attributes can be bundled this way. For example, point lights, spotlights, area lights, and directional lights, which are represented by attributes, share the attribute category "light-source".

The attribute table distinguishes between *mono attributes* and *poly attributes*. Only the most recently stored mono attribute of a category is considered to be active. For each category of mono attributes the rendering context allocates a separate attribute stack. Poly attributes of a specific category can be active in any number. For each category of poly attribute, the rendering context allocates a unique set. An attribute is either a mono attribute or a poly attribute. The most recently stored mono attributes of each category and the currently included poly attributes of each category at a given point in time represent the *current context* of an engine.

The *handler table* allocates for each (service, target) pair a handler stack (Figure 5). Handlers are treated like mono attributes except that two parts, the service and the target determine, their category. Conceptually, the handler table is a two-dimensional array of stacks, addressed by service and target identifiers, but not all array elements are actually used. In general, the handler table of an engine is set up at construction time and can be modified later, i.e., engines can be reconfigured at run-time.

The rendering context allocates appropriate containers automatically if a new attribute category, service or target is detected. The generic management of attributes and handlers is essential for the generic rendering system, because otherwise we would have to freeze the set of supported attribute categories and handlers which would restrict the modeling of individual features of a wrapped rendering system or rendering technique.

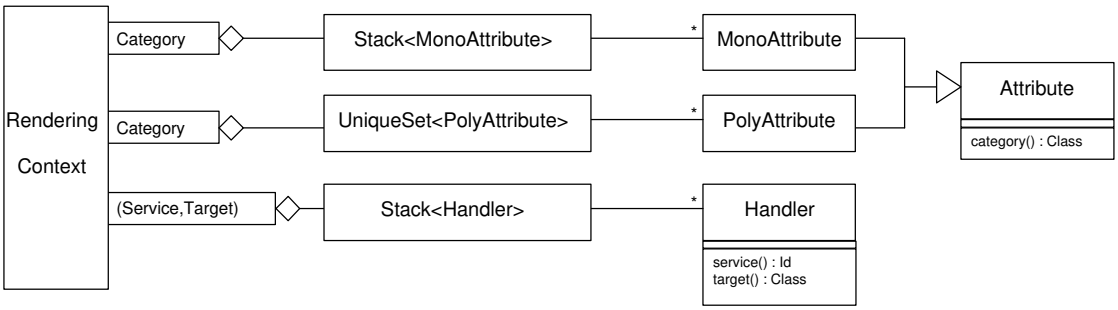


Figure 5. Class model of the rendering context.

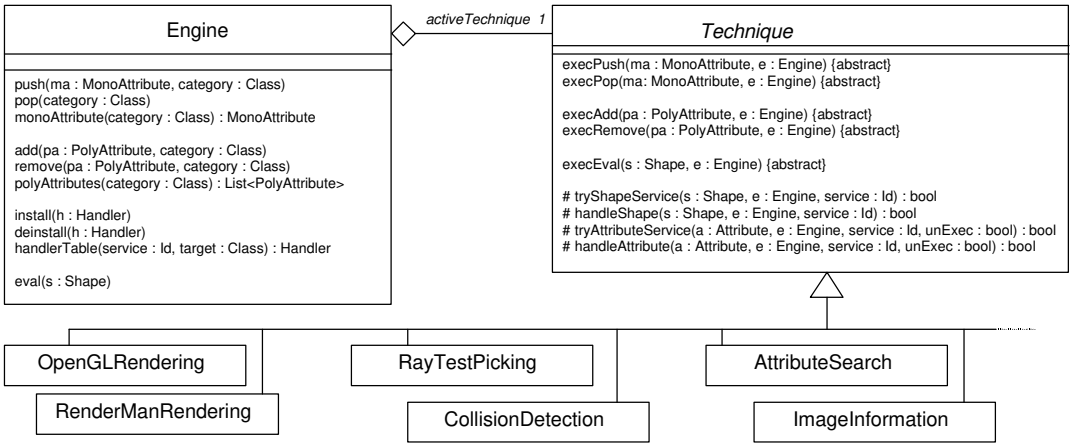


Figure 6. Class interfaces of engines and techniques.

OpenGL whose architecture is based on a state machine has motivated the design for the rendering context. A state machine provides the finest possible control over rendering attributes because it allows an application to modify exactly those attribute values which are different from the current attribute values and to store/restore attribute values temporarily using stacks. We have extended this concept to an object-based, generic state machine: The rendering context stores all state-related attribute objects and handler objects, and automatically creates stacks or sets for new attribute categories.

The generic rendering system models shapes and attributes independently and uses the rendering context for their association. The concrete handler used to evaluate a shape decides which of the attributes of the current context to deploy. As a consequence, shapes need not store attribute values, i.e., they are small in terms of memory usage. The lightweight design [5] ensures that shape objects and attribute objects are as small as possible, and that they can be used in large numbers and implemented efficiently [20].

5.2 Evaluation Strategies

The engine does not implement any strategy for evaluating rendering components; this is modeled separately by techniques. The interface of the engine class (Figure 6) serves as primary interface to the generic rendering system. It can take advantage of the coarse subdivision of rendering components in shapes, mono attributes, poly attributes, handlers, and techniques to reduce the number of methods: There are only methods for pushing and popping mono attributes, for adding and removing poly attributes, installing and deinstalling handlers, and evaluating shapes. Each engine has an active technique; it can be replaced or temporarily substituted at run-time.

We distinguish two kinds of techniques: techniques that depend on the low-level rendering and techniques that are independent from a concrete low-level rendering system.

- A *rendering technique* synthesizes images for a concrete low-level rendering system. It maps attributes and shapes to appropriate constructs of a low-level rendering system, e.g., OpenGL or RenderMan.
- The *ray-picking technique* determines object-ray intersections using ray-tracing. It is used, for example, to determine which object has been picked by the user.
- The *collision-detection technique* determines which objects collide. It skips most attributes and checks for collision only those shapes that are tagged to be relevant for collision detection.
- The *attribute-search technique* records the current context for a given shape, i.e., it records which attributes would have been actually applied to that shape.

- The *image-information technique* calculates the position and extension of a given shape in the view plane.

The non-rendering techniques do not synthesize images and therefore do not depend on any low-level rendering system. In general, they can skip most attributes and shapes that are not relevant to them.

If an attribute is sent to an engine, the engine stores the attribute in its rendering context. Then, the engine delegates the evaluation of the attribute to its active technique. The technique will look up an appropriate handler. For example, if a material attribute is sent to an engine, it is stored in the material attribute stack of the rendering context. Then, the technique searches for a handler with the service "attribute deployment" and the target "material"; the handler, if found, could map the attribute to equivalent commands of a low-level rendering system.

If a shape is sent to an engine, the engine delegates the evaluation to its active technique, which searches for a suitable handler and delegates the evaluation to it. For example, the OpenGL rendering technique searches for handlers with the service "OpenGL painting" and the shape class as target.

If a technique cannot find a handler for evaluating an attribute or shape, it tries to find simplifiers for that attribute or shape. If no simplifiers are available, we repeat the search using the parent class of the object to be evaluated as target for the handler table. If no handler can be found at all, the application-specific error handling undertakes the task. The technique base class *Technique* provides methods that implement this scheme for handler look-up. For shapes, the method *handleShape* implements the recursive look-up for services. The methods for attributes are implemented analogously; we have to distinguish, however, between activation of an attribute (e.g., push or add) and deactivation (e.g., pop or remove). An implementation of the shape related methods is outlined below:

```
void OpenGLRendering::execEval(Shape s, Engine e) {
  if(handleShape(s, e, OPENGLPAINTING)==false) {
    error handling for missing service
  }
}

bool Technique::handleShape(Shape s, Engine e, Id service) {
  Class target = s.Class();
  while (target!=NULL) {
    Handler h = e.context().handlerTable(service, target);
    if(h!=NULL) {
      h.exec(s,e);
      return true;
    }

    h = e.context().handlerTable(SIMPLIFICATION, target);
    if(h!=NULL) {
      h.exec(s,e);
      return true;
    }

    // no success, find handlers for parent class
    target = target.parentClass();
  }

  // no suitables handlers found
  return false;
}
```

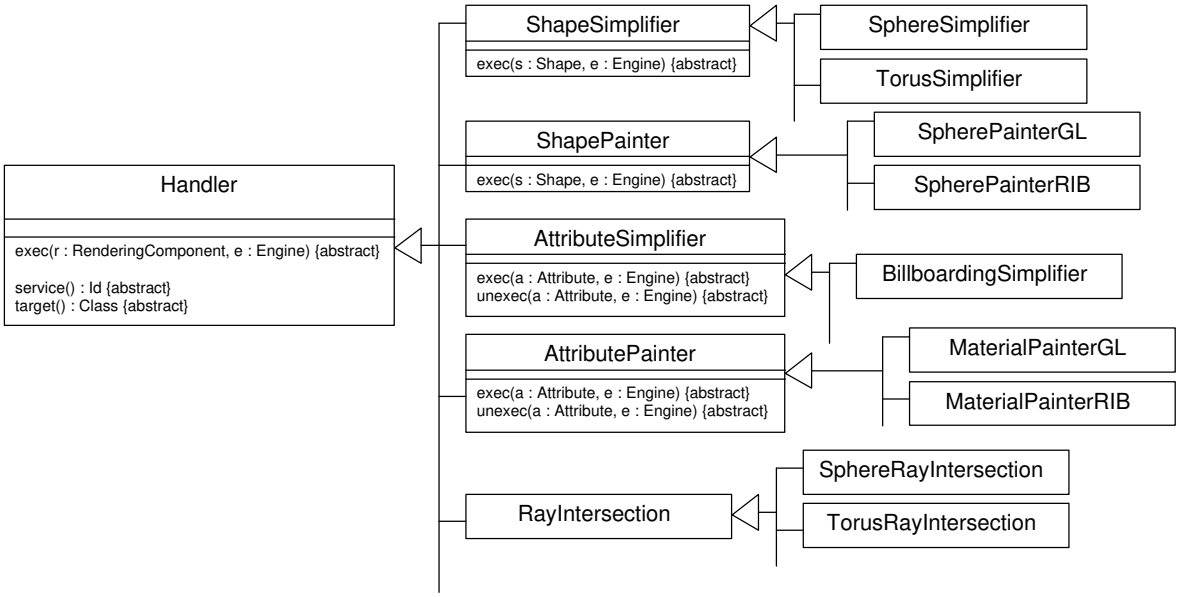


Figure 7. Class model of the handler hierarchy.

5.3 Integrating Rendering Algorithms

Handler classes implement any kind of rendering algorithm; they complement the implementation of shapes and attributes. Consequently, the engine, shape, and attribute classes are kept simple, leading to *lightweight rendering components.* The most important services defined by the generic rendering system, whose class model is depicted in Figure 7, include:

- *Shape Simplification*: A shape simplifier is responsible for decomposing a complex shape into a collection of less complex rendering components. This allows low-level rendering systems to draw complex shapes without having to support that type of geometry. For example, a torus simplifier may convert a torus into a triangle mesh. If a technique deploys a simplifier, it calls the engine recursively for the resulting rendering components.
- *Shape Painting*: A shape painter is responsible for mapping a shape to constructs of a low-level rendering system. The shape painter may consider the current context to decide how to map the shape. For example, the OpenGL painter for triangle meshes outputs OpenGL triangle lists; if the context contains a texture, it activates the texture and sends additional texture coordinates for each triangle. For one shape class, handlers for different low-level rendering systems (and versions) may be provided, e.g., shape painters for OpenGL 1.1 and OpenGL 1.2.
- *Attribute Simplification*: An attribute simplifier is responsible for decomposing a complex attribute into a collection of less complex attributes. An attribute that is specific to one low-level rendering system can be mapped to appropriate attributes of another rendering system. For example, the plastic attribute used for the RenderMan system could be mapped to a color attribute and a material attribute used for OpenGL.
- *Attribute Painting*: An attribute painter is responsible for mapping an attribute to appropriate constructs of a low-level rendering system. Like a shape painter, an attribute painter is specific to a low-level rendering system. For example, the OpenGL color painter modifies the color of the current OpenGL context.
- *Ray Intersection*: A ray intersector intersects a ray with shapes. If for a given shape as target the rendering context does not contain a ray intersector, the shape is simplified, and the ray intersection is performed recursively on the result of the simplification.

Normally, handlers and techniques are not visible to developers using the generic rendering system, because the engine constructor sets up the rendering context with appropriate handlers. If developers want to provide customized (e.g., new or optimized) handlers, they can implement new handler classes and register them in the rendering context of an engine at any time. The registration can be automated, i.e., each handler class can declare for which engine classes it will become a default handler.

5.4 Using the Engine Interface

The following examples[1] exemplify the evaluation of rendering components by engines. In the first example, a torus is rendered, associated with a material attribute and transformed by a rotation attribute. In the case of an OpenGL rendering technique, the *eval* method would use a simplifier handler to decompose the torus into polygons, and render the resulting polygons.

```
void example(Engine e) {
  Material mat = new Material(...);
  Rotation rot = new Rotation(...);
  Torus ts = new Torus(...);

  e.push(mat, mat.category());
  e.push(rot, rot.category());
  e.eval(ts);
  e.pop(mat.category());
  e.pop(rot.category());
}
```

The next example demonstrates how specific rendering algorithms can be temporarily associated with shapes. Assume, we develop a specialized OpenGL painter for torus shapes that provides a more efficient implementation than the simplifier approach of the previous example. Only an OpenGL rendering technique uses that painter, otherwise it has no effect, i.e., the command sequence remains generic.

```
void example(Engine e) {
  Torus ts = new Torus(...);
  TorusPainterOpenGL tspainter = new TorusPainterGL();

  e.install(tspainter);
  e.eval(ts);
  e.deinstall(tspainter);
}
```

The third example demonstrates attribute simplification. It uses a billboarding transformation, which transforms the current model-view matrix such that a specified axis (e.g., the z axis) points towards the camera. The billboarding transformation cannot be calculated in advance because it depends on the current model-view matrix. The calculation is performed by the billboarding simplifier, which has access (like all handlers) to the engine's context. The billboarding simplifier produces a sequence of elementary transformations, which, again, are sent to the engine.

```
void BillboardingSimplifier::exec(b : Billboarding, e : Engine) {
  List<Transformation> T_b = new List<Transformation>();
  T_b ← createTransformations(b, e);
  for each t in T_b {
    e.push(t, t.category());
```

[1] The examples are given in a notation similar to C++ and Java. Note that memory management issues are ignored.

```
  }
}

void BillboardingSimplifier::unexec(b : Billboarding, e : Engine) {
  for each t in T_b {
    e.pop(t.category());
  }
}
```

The billboarding attribute, however, can be used like an elementary attribute from a developer's point of view.

```
void example(Engine e) {
  Billboarding bb = new Billboarding(...);
  Torus ts = new Torus(...);

  e.push(bb, bb.category());
  e.eval(ts);
  e.pop(bb.category());
}
```

6 HIERARCHICAL MODELING OF RENDERING COMPONENTS

The generic rendering system supports the hierarchical modeling of rendering components by scene graphs. It can be implemented in a straightforward manner: The nodes contain rendering components, engines traverse the graph, and nodes send the rendering components to the engine. We briefly outline a scene graph implementation. Based on the generic rendering system, different schemes for scene modeling can be implemented as well. For a discussion of hierarchical graphical scenes, see [3].

The scene graph of the generic rendering system is composed of scene graph nodes and rendering objects (Figure 8). Scene graph nodes organize rendering components in a hierarchical manner; they can also generate and constrain rendering components.

We can distinguish two types of scene graph traversals: evaluation and inspection. The *evaluation* traversal uses an engine to interpret rendering components contained in scene graph nodes (e.g., for image synthesis). In contrast, the *inspection* traversal only explores the scene graph, its contents and graph structure (e.g., for scene graph storage). Both traversals are implemented based on the visitor design pattern.

6.1 Generic Scene Graph Node

A scene graph node stores rendering components and references to subgraphs in a single, inhomogeneous list. During evaluation, scene graph nodes send shapes and attributes to the current engine and initiate the recursive traversal of subgraphs. The implementation below outlines the node class:

```
class Node {
private: List<Object> contentObjects;
public:
  void evaluate(e : Engine) {
    unapply(apply(contentObjects,e),e);
  }
  void inspect(v : Visitor) {
    for each c in contentObjects do {
      v.explore(c)
      if(c is a node) { c.inspect(v); }
    }
  }
};

Stack apply(L : List<Object>, e : Engine) {
  S : Stack = {};
  for each c in L {
    if(c is a node) { c.evaluate(e); }
    else if(c is a shape) { e.eval(c); }
    else {
      S.push(c);
      if(c is a mono attribute) { e.push(c, c.category()); }
```

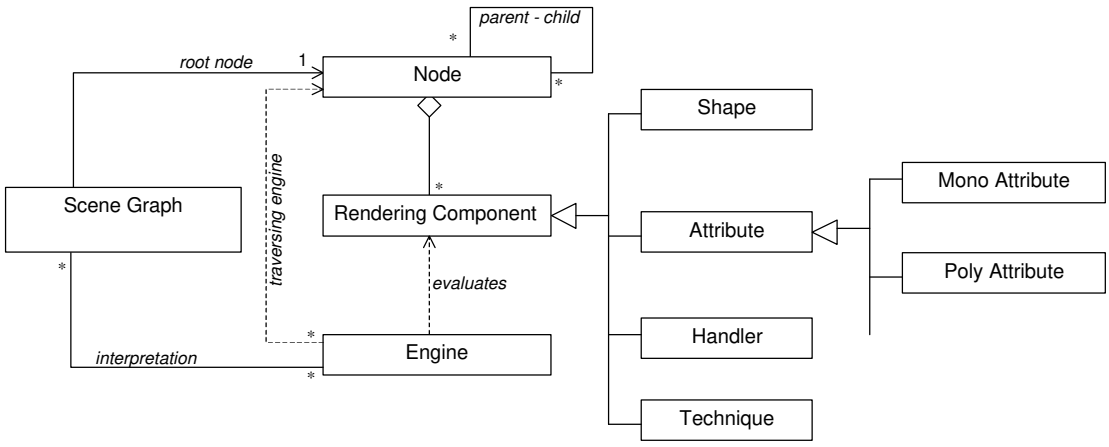


Figure 8. Class diagram of the scene graph of the generic rendering system.

```
      else if(c is a poly attribute) { e.add(c, c.category()); }
      else if(c is a handler) { e.install(c); }
    }
  }
  return S;
}

void unapply(S : Stack<Object>, e : Engine) {
  while (S not empty) {
    c ← S.pop()
    if(c is a mono attribute) e.pop(c.category())
    else if(c is a poly attribute) e.remove(c, c.category())
    else if(c is a handler) e.deinstall(c)
  }
}
```

During evaluation, scene graph nodes formulate a "rendering micro program" (Figure 9). Mono attributes are pushed to (popped from) the engine's context; poly attributes are included in (excluded from) the collection they belong to; and handlers are installed (de-installed) in the handler table of the context. The node class is generic, because all types of content objects can be arbitrarily mixed, which leads to compact scene specifications. In any case, the attributes contained in a node affect only its children and never its sibling nodes.

6.2 Interfacing Low-Level Rendering Systems

The scene graph of the generic rendering system cooperates with different rendering systems. For each supported low-level rendering system, the generic rendering system implements the handlers for built-in shapes and attributes, the attributes for system-specific features, and a specialized rendering engine.

Attribute types differ to a high degree among rendering systems. Therefore, the scene graph permits to store any attribute type. Attributes are not evaluated unless a suitable attribute painter or simplifier is installed. This way, attributes not applicable to a rendering system do not harm. The generic rendering system defines a small collection of standard attributes (e.g., appearance and transformation attributes), and provides specialized attributes for each supported low-level rendering system. For example, OpenGL-specific attributes cover most of OpenGL's functionality. For RenderMan, a shader attribute interfaces compiled RenderMan shader files.

In particular, renderer-specific attributes, which are included as regular attributes in scene graphs, facilitate the production of high-quality animations: manual post-processing of exported scene descriptions is no longer necessary because all details of the target rendering system can be expressed by the generic rendering system. For all built-in attributes, default conversions are available to map the attributes reasonably (e.g., materials and light sources).

In the sample scene graph in Figure 10, RenderMan attributes (*RenderManShader*) are specified. The scene displays a Wavefront object (*WaveFrontObject*), rendered by the OpenGL-specific shape painter (*WaveFrontPainterGL*) or simplified (*WaveFrontSimplifier*) in the case of all other rendering systems. The *WaveFrontObject* also exemplifies higher-level shapes, which the generic rendering systems allows us to specify: A scene object in WaveFront format includes geometry data and may

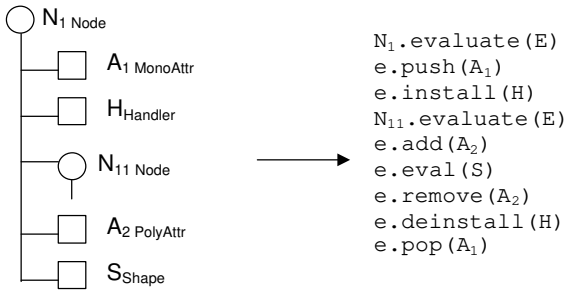


Figure 9. A scene graph node and its content objects (left). Resulting engine microprogram (right).

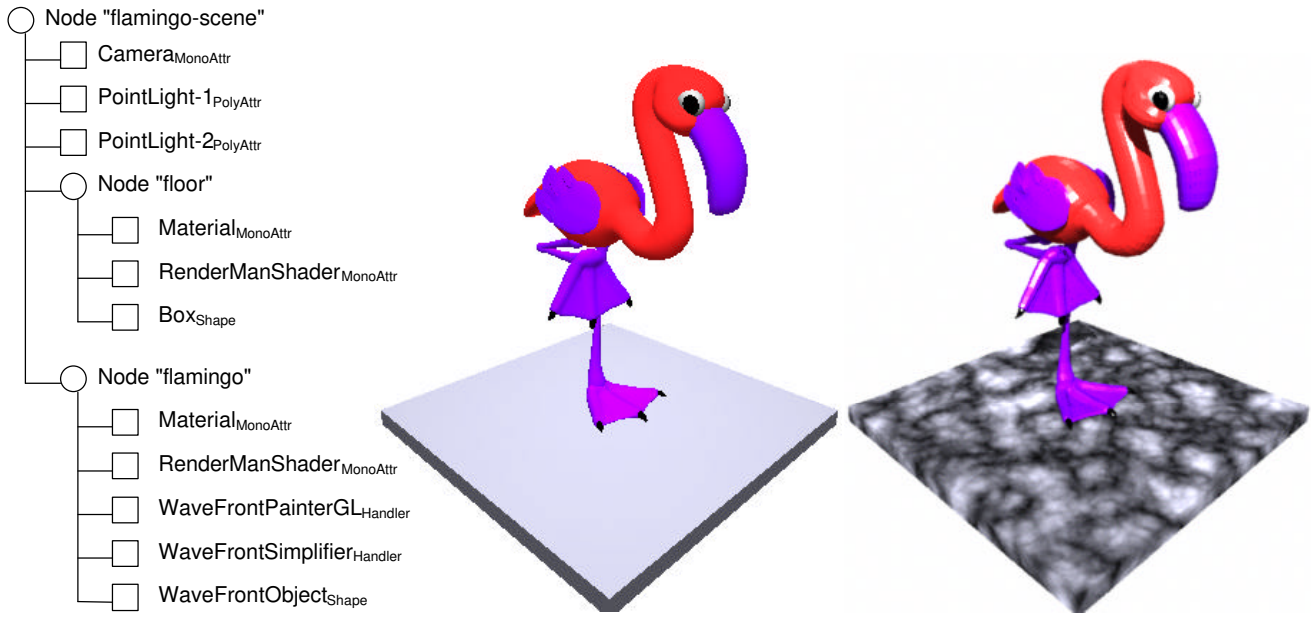


Figure 10. Example of a scene graph, rendered by OpenGL and RenderMan.

assign each geometry part different graphics attributes. Simplification handlers are able to decompose a given shape into a sequence of rendering objects that not only contains shapes but also attributes.

6.3 Characteristics of the Scene Graph

The following design principles characterize the scene graph of the generic rendering system:

- *Strict separation of shapes and attributes* similar to Vision [31]. Furthermore, there is no restriction for the types of attributes – engines provide a generic rendering context. In addition, shapes and attributes can depend on the current context (e.g., billboarding simplification). This allows for intelligent, automated shapes and attributes.
- *Strict separation of declaration and implementation*. Handlers encapsulate all kinds of algorithms applied to shapes and attributes. Therefore, algorithms can be substituted in order to introduce application-specific implementations or to optimize the default implementation. In addition, handlers may vary from subgraph to subgraph, enabling the application of different algorithms to the same class of shape or attribute.
- *Separation of structure and content*. Scene graphs provide structure; rendering objects provide contents. Both class categories can be extended independently.
- *Scene graphs are understood as parameterized scene content specifications.* A scene graph represents a template that is instantiated if a concrete engine interprets the scene graph. The technique associated with the engine and the handlers deployed by the technique determine the kind of rendering.

7 INTEGRATING SHAPE TYPES AND SHAPE DATA

The generic rendering system supports the definition and seamless integration of application-specific shape types and application data.

7.1 Integrating Shape Types

New shape types may become necessary to support new modeling techniques or to support application-specific shapes. If new shape types cannot be integrated, i.e., if the collection of built-in shape types is not extensible, an application would be forced to convert these shapes to built-in shapes. Consider for example an OpenGL application working with large numbers of 3D arrows. We can extend the generic rendering system by a new arrow shape class and provide an OpenGL arrow painter, possibly optimized for the application's purposes. If we have to convert arrows to built-in shapes by combining a cone shape and a cylinder shape then two problems arise: (1) The application stores redundant information, i.e., for each arrow a cone shape and a cylinder shape. (2) The arrow semantics is lost in the rendering system. The semantics, however, may be useful to optimize rendering and intersection algorithms.

New shape types can be integrated into the generic rendering system by shape classes and corresponding handler classes. A minimal implementation consists of the shape class and a simplifier class, which converts shapes of the new type into collections of built-in shapes. A more sophisticated implementation of a shape type could provide shape painters for low-level rendering systems and a ray-intersection handler. The design pattern for shape types is depicted in Figure 11.

7.2 Integrating Shape Data

Shape objects are typically defined by data arrays. For example, polygonal shapes may require a set of vertex coordinates, vertex normals, vertex colors, and vertex texture coordinates. The assumption that the data of a shape is stored in an array has several drawbacks: applications cannot choose the data structure in which they want to represent the shape data (e.g., array or list), and one has to pay the costs for converting or copying the data.

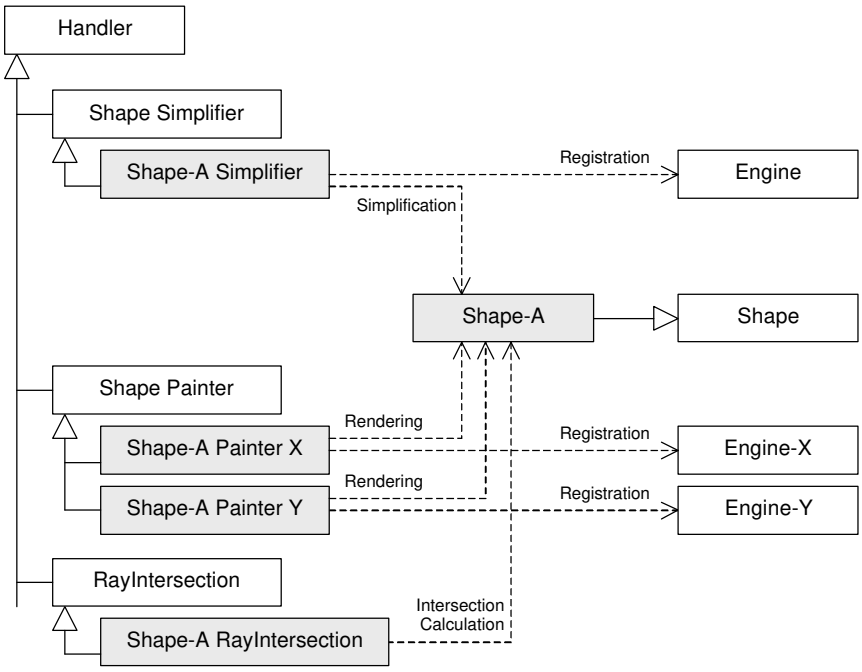


Figure 11. Design pattern for shape types.

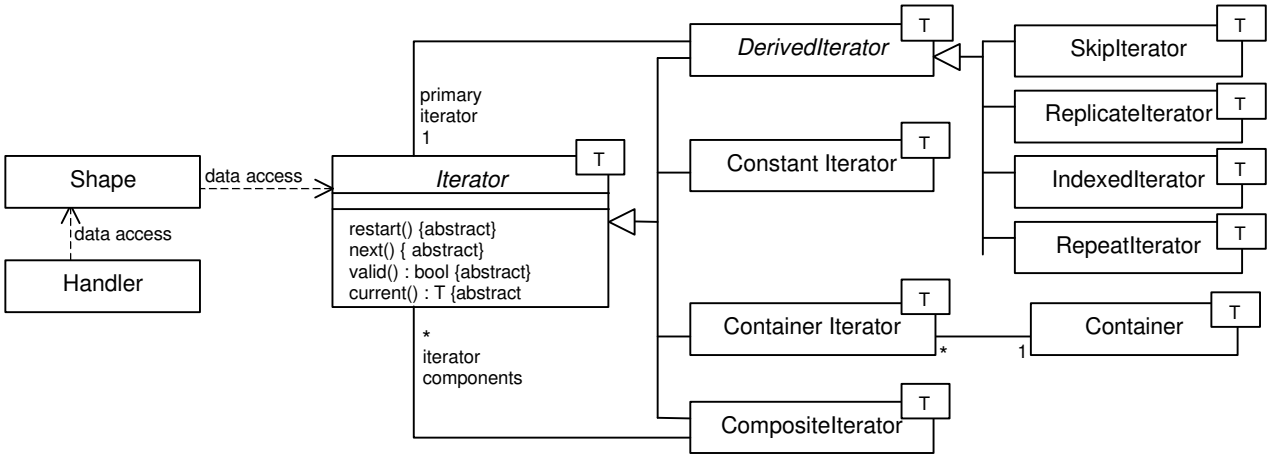


Figure 12. Class hierarchy of iterators.

7.2.1 Iterators for Accessing Data

Shape classes of the generic rendering system use iterators to bind data to shapes. An *iterator* is an object that provides sequential access to data elements. It hides the internal representation of the data, which either can be contained in data storage or be functionally generated. Since shapes and iterators know nothing about each others implementation, the data of a shape can be stored in different ways without affecting the interface of the shape class, and application-specific data structures can be attached to shapes by providing suitable iterators, i.e., algorithms operate directly on shape data contained in application data structures. Since no data are copied into a shape object integrity problems cannot arise. Furthermore several iterators may supply the same data to different shape objects. This approach has been motivated by the Standard Template Library [24].

The relationship between shapes, handlers, and iterators is depicted in Figure 12. The base iterator class, conceptually a template class with *T* as the type of data, embodies methods for resetting the iterator to the first element, stepping to the next element, testing for more elements, and accessing the current element. The generic rendering system defines iterators for all common container classes such as arrays, lists, or queues. This ensures that the iterator approach can be used as simple as possible for standard data containers.

Typical shape classes define their geometric data by means of iterators. For example, a polygon shape could require three iterators (Figure 13) that define vertex coordinates, vertex normals, and vertex colors (more iterators could be used, for example, for vertex texture coordinates and edge flags). When the polygon is sent to an engine the selected handler, e.g. a polygon painter, will use the iterators to inquire the data.

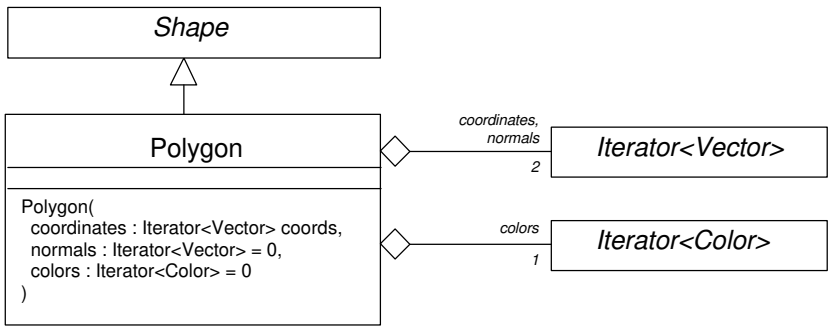


Figure 13. Iterator-based data integration for generic rendering components.

7.2.2 High-Level Iterators

Based on the iterator class and its use for connecting shapes with data sources, several high-level iterator classes enable applications to efficiently represent data sequences with certain characteristics:

- A *constant iterator* generates a constant sequence, i.e. a sequence of identical values. It is used to reduce the memory requirements.
- A *replicate iterator* generates a sequence which results from replicating each data value of another iterator a given number of times, e.g. it generates "aaabbbcccddd..." from the given sequence "abcd...".
- A *repeat iterator* generates a sequence of data values that results from replicating a segment of a given iterator, e.g. it generates "bcdbcdbcdbcd..." from "abcde..." for the interval [1,3].
- A *skip iterator* generates a sequence of data values that results from skipping certain data elements of another iterator sequence, e.g. it generates "abdeghj..." from "abcdefghij..." when skipping every third element.
- An *indexed iterator* represents an indexed sequence of data values specified by a sequence of indices and a sequence of source data values, e.g. it generates "acgh..." from "1,3,7,8,..." and "abcdefghijklmno...".
- A *composite iterator* represents a sequence that is obtained by combining two or more iterators sequentially.

For example, consider a polygon specified by a list of *N* vertex coordinates with alternating binary coloring (blue and white), and constant vertex normals (Figure 14). The polygon shape constructor expects iterators for vertex coordinates, vertex normals, and vertex colors. The vertex iterator returns the data stored in the vertex array, the normal iterator builds a constant sequence of *N* vectors, and the color iterator alternately repeats the two colors *N/2+1* times.

The iterator concept does not have a negative impact on the performance compared to explicitly storing shape data in shape objects. For objects whose geometry is static, display lists are used for real-time rendering. For objects with dynamic geometry data, the iterator implies only a small run-time overhead. If a time-critical painter or simplifier for a specific real-time rendering system determines from the run-time type information that the iterator takes its source data from a memory array, it can circumvent the iterator traversal and directly access the memory of the data (e.g., using vertex arrays). This approach leads to an efficient and flexible scheme for associating shapes with data. In general in computer graphics the functional paradigm provides more flexibility compared to declarative approaches [13].

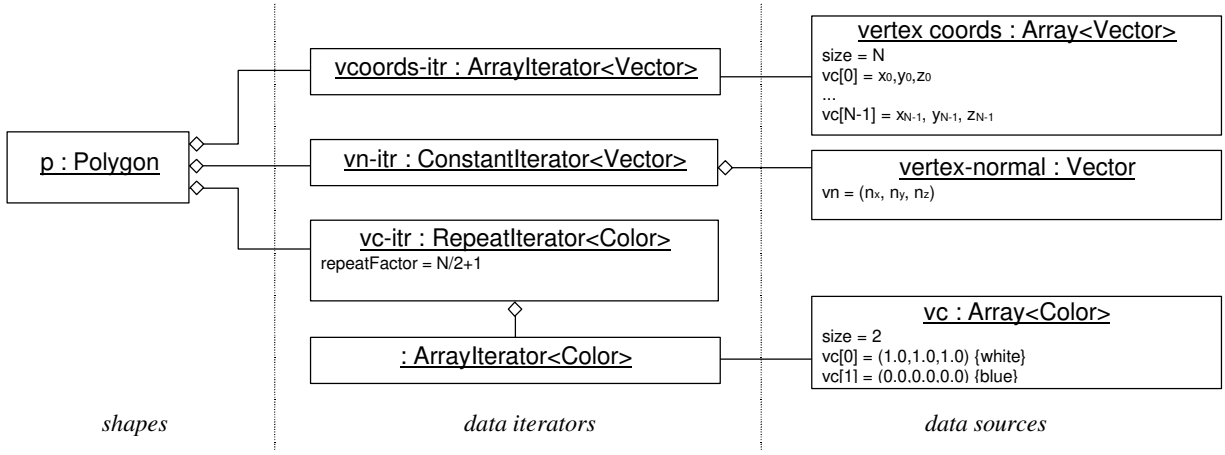


Figure 14. Sample configuration of a shape, data iterators, and data sources.

8 ATTRIBUTE DESIGN

Attributes include all kinds of modifiers that control or specify the evaluation of shapes in a target medium. The generic rendering system classifies attributes as follows:

- *Surface shading attributes* specify the shading of surfaces, and include color, material, texture and styles for facets, lines, and points.
- *Scene attributes* specify the rendering of the environment and include fog, depth-cueing, anti-aliasing, and additional, global rendering properties.
- *Light source attributes* represent virtual light sources, which are distinguished in distant lights, point lights, spotlights, and area lights.
- *Geometry attributes* modify or manipulate the geometry of shapes. Transformations are specified by 4×4 matrices. Specialized transformations include perspective projection, orthogonal projection, scaling, translation, rotation, reflection, direction-of-flight transformation, polar transformation, billboard transformation, etc.). Geometry attributes include also clipping planes, level-of-detail attributes, and tessellation attributes.

In general, rendering systems based on a local illumination model support similar attributes due to the same shading techniques; rendering systems based on a global illumination model support more complex attributes with respect to reflection, transmission, and emission of light. Therefore, we cannot define one closed set of attribute types, which would eliminate the support for all non-common (i.e., unique) features of low-level rendering systems. Consequently, the generic rendering system does not limit the number of attribute categories to enable renderer-specific and application-specific attribute classes. For example, VRS provides Radiance and RenderMan attribute types to make the specific shading and lighting features of these rendering systems available.

The generic rendering system defines a core set of attribute types that are general enough to produce reasonable results for different low-level rendering systems. The core attribute set includes colors, geometric transformations, clipping planes, and light sources. The degree to which other attributes can be evaluated depends on the capabilities of the low-level rendering system. This approach allows the generic rendering system to produce images that reflect reasonably well the specified attributes without having to modify the application code when exchanging the rendering system.

The attributes sent to an engine can be evaluated in two different ways. Either a shape painter may directly inquire the values of all relevant current attributes and use these values to set up the context of the low-level rendering system, or an attribute painter, which is called when an attribute is pushed to or popped from the engine, sets up the context of the low-level rendering system. In the case of OpenGL, attribute painters handle most attributes because there is a direct correspondence between attributes and OpenGL context variables.

Attributes are completely decoupled from shape management, i.e., engines process attributes independently from shapes. A handler may use the active elements of each attribute category, but attributes are not directly bound to shapes. This ensures that new attribute types can extend the generic rendering system without having to modify the shape classes. The painters may migrate and consider new attribute types while the shape interfaces remain stable.

9 MULTI-PASS RENDERING TECHNIQUES

Multi-pass rendering refers to a process in which scene objects are evaluated two or more times to generate a single image. In particular, real-time rendering uses multi-pass rendering [16] to achieve high-quality illumination and shading effects such as reflections [18], bump mapping [17], shadows [16], or to apply programmable shading techniques [25] as well as to implement image-based geometric modeling techniques such as constructive solid geometry [41].

Consider a 3D scene described by a set of shapes and attributes, which are organized in a directed, acyclic scene graph. During the traversal of the scene description, a sequence of rendering components is generated and sent to an engine. Multi-pass rendering techniques need to process the sequence of rendering components multiple times. In the generic rendering system, multi-pass rendering can be defined as part of a technique.

9.1 Control of Rendering Passes

A technique encapsulates a strategy for processing a sequence of rendering components. A technique may evaluate such a sequence only once or multiple times. The methods used to control rendering passes are depicted in Figure 15. The *start* method initializes the run of a technique (e.g., clears framebuffer resources); the *needsPass* method returns whether there are still passes to process; the *preparePass* method checks if the current pass has to be processed based on the context of the engine; the *finishPass* method terminates a single pass; the *nextPass* method sets up the technique for the next pass; and the *stop* method terminates the run of a technique.

The technique decides how many times the sequence of rendering components has to be traversed. For this, scene graph nodes use techniques to control the evaluation of their content objects. That is, if a scene graph node contains a technique, the node hands over local control of the traversal to this technique. A prototypical algorithm for processing a sequence of rendering components with a technique *t* for an engine *e* is shown in Figure 15 (right).

The technique also specifies how to evaluate shapes and attributes. In general, the *exec* methods of a technique look up and execute suitable handlers. A concrete technique class can optimize the execution, for example, by ignoring shape and attribute classes that are not relevant to the current pass.

In general, rendering techniques require one pass. For real-time rendering systems such as OpenGL, the rendering technique requires typically two passes. The first pass evaluates opaque objects only, and the second pass evaluates any transparent objects detected during the first pass. For scene anti-aliasing additional passes may become necessary.

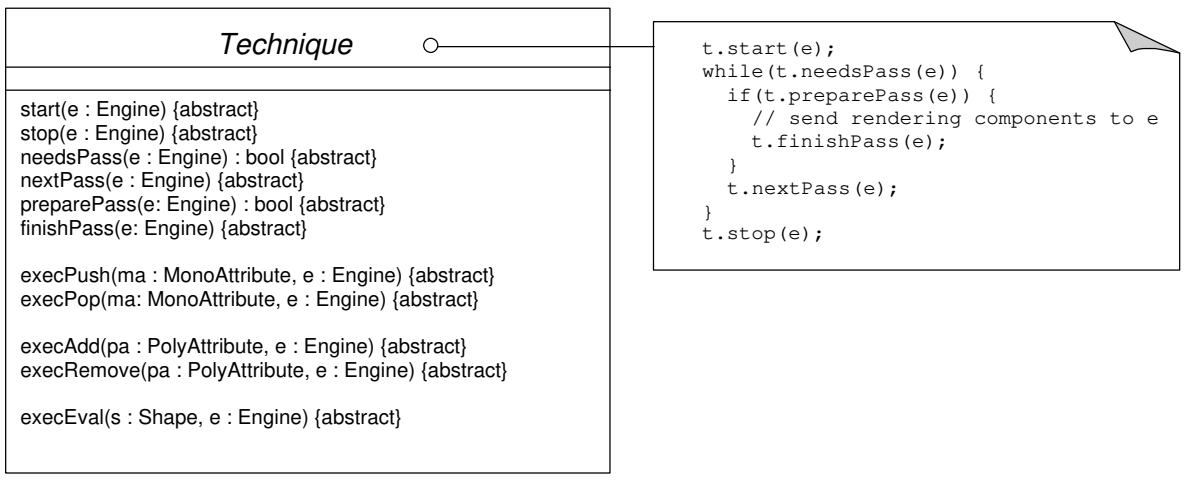


Figure 15. Technique interface, extended by pass-control methods for support of multi-pass rendering.

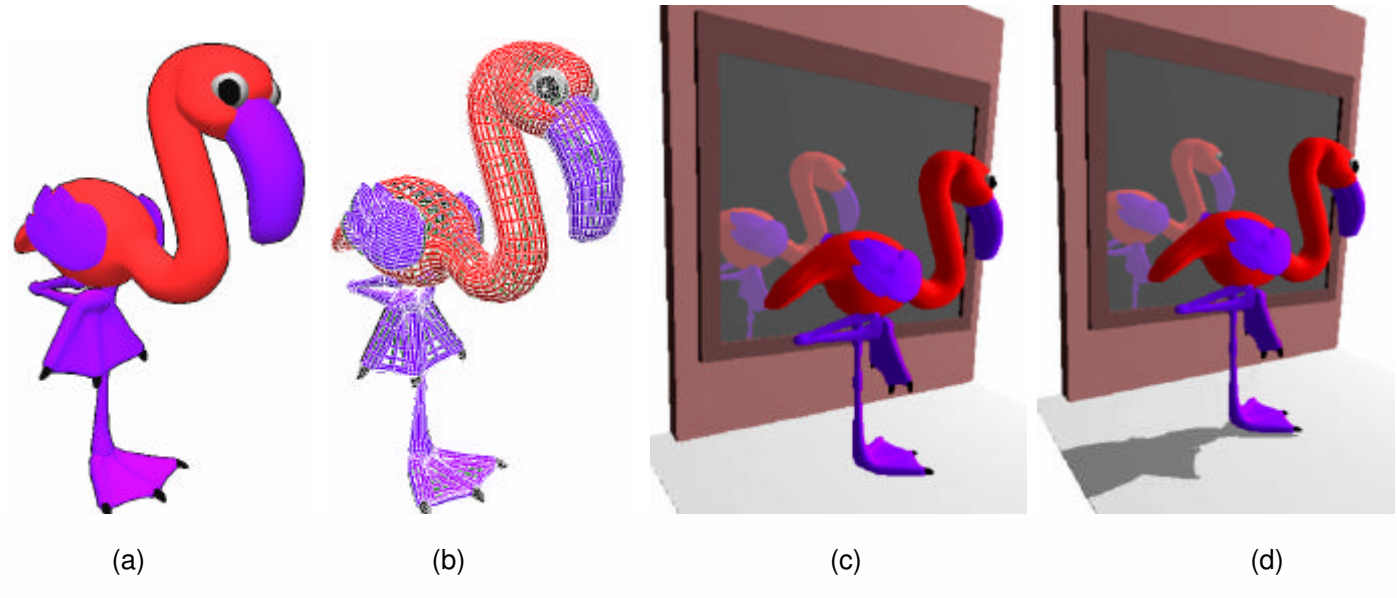


Figure 16. (a) Rendering with enhanced silhouettes. (b) Haloed wire-frame rendering. (c) Rendering with reflection technique. (d) Rendering with reflection and shadow techniques.

Non-photorealistic rendering techniques can be integrated seamlessly into the generic rendering system by encapsulating them in specialized rendering techniques. Such a technique can be used to extract image information into G-buffers [28] such as surface normal information, object identifiers, and surface parameterization, which are used later for generating or post-processing an image.

9.2 Implementing Advanced Graphics Programming Techniques

Techniques permit a straightforward implementation of the so-called advanced graphics programming techniques using OpenGL [21]. These techniques, which are mostly implemented by multi-pass rendering techniques, lead to a higher degree of scene realism and to new rendering styles. The generic rendering system handles these techniques as high-level attributes, which can be inserted as content objects in scene graphs and then apply to certain shapes or parts of a scene. Examples of advanced OpenGL rendering techniques include:

- The *surface-style technique* can draw a shape in enhanced wire-frame styles (e.g., with additional silhouette or as haloe line drawing). Basically, this technique uses the stencil buffer and accumulation buffer of OpenGL [21].
- The *reflection technique* implements reflections on planar surfaces. This technique uses the stencil buffer to identify mirroring surfaces in the image, and renders the mirrored scene in an additional pass [18] in these areas.
- The *shadow technique* implements shadows based on the shadow-volume algorithm. In the first pass, it constructs the shadow volume for a given set of shadowing objects, and using the stencil buffer it applies the shadow to the non-shadow objects and the shadowed objects in the second and third pass, respectively [18].
- The *Phong-highlight* technique implements texture-based highlight patterns by a two-pass rendering algorithm. In the first pass, the rendering components are evaluated without specular lighting; in the second pass, the rendering components are evaluated in the blending rendering mode and get textured by the highlight pattern.

Figure 16 illustrates the effects of the surface-style, reflection, and shadow techniques. More techniques such as bump-mapping and rendering of constructive solid geometry expressions can be implemented as well. The concept of techniques gives developers full access to advanced capabilities of

a low-level rendering system through a transparent, object-oriented design. In particular, techniques are well suited to wrap multi-pass rendering techniques of OpenGL in reusable building blocks.

9.3 Using Techniques as Attributes

Many techniques are used like attributes, i.e., they define visual, geometric, and conceptual properties of shapes. Techniques can be evaluated together with regular attributes such that from the developer's point of view there is no distinction between them. A scheme for the recursive evaluation of a sequence of rendering components *C* by an engine *E* including techniques is outlined below. It replaces the *apply/unapply* procedure defined for the node class of Section 6.

```
void Node::evaluate(Engine e) {
  // content objects are stored in arrays
  int to = apply(e, contentObjects, 0);
  unapply(e, contentObjects, 0, to)
}

int apply(Engine e, Array C, int from) {
  int N = C.size();
  for(int i=from; i<N; i++) {
    RenderingComponent c = C[i];
    if(c is a node) { c.evaluate(e); }
    else if(c is a shape) { e.eval(c); }
    else if(c is a mono attribute) { e.push(c, c.category()); }
    else if(c is a poly attribute) { e.add(c, c.category()); }
    else if(c is a handler)        { e.install(c); }
    else if(c is a technique) {
        c.start(e);
        while(c.needsPass(e)) {
          if(c.preparePass(e)) {
            int to = apply(e, C, i+1);  // recursive eval
            unapply(e, C, i+1, to);
            c.finishPass(e);
          }
          c.nextPass();
        }
        c.stop(e);
        return i-1;
    }
  }
  return N-1;
}

void unapply(Engine e, Array c, int from, int to) {
  for(int i=to; i>=from; i--) {
    RenderingComponent c = C[i];
    if(c is a handler) { e.deinstall(c); }
    else if(c is a mono attribute) { e.pop(c.category()); }
    else if(c is a poly attribute) { e.remove(c.category()); }
  }
}
```

For example, consider a sequence of rendering components, which are contained in a scene node N, consisting of a texture attribute A_0, a Phong-highlight technique A_1 and a surface style technique A_2, and a shape *S*. The sequence $C=\{A_0, A_1, A_2, S\}$ defined by the node component objects is evaluated by a nested multi-pass traversal.

```
Node N = new Node();
N.add(A0 = new Texture(...));
N.add(A1 = new PhongHighlight(...));
N.add(A2 = new SurfaceStyle(...));
N.add(S = new Polygon(...));
N.eval(engine);
```

10 APPLICATIONS AND EXPERIENCE

The *Virtual Rendering System* (VRS) is a proof-of-concept implementation of the generic rendering system. It has been implemented as a portable C++ library and provides as built-in features a collection of shapes, shape simplifiers, shape ray-intersectors, attributes, and techniques.

The easiest way to integrate a low-level rendering system into VRS is to provide the shape painter class for triangle sets and a core set of attribute painters. VRS can map all pre-defined shapes to triangle sets. A typical integration will also provide shape painters for those shape types directly supported by the low-level rendering system.

10.1 Wrapping OpenGL

The VRS adapter for OpenGL provides a collection of OpenGL-specific attributes, handlers and techniques, e.g., attributes for controlling the color buffer, depth buffer, stencil buffer, scissor test, stencil test, alpha test, polygon offset and texture features.

For all polygon-based shape classes and standard attribute classes, VRS provides OpenGL painter classes to ensure optimal performance. Array iterators used by these painters access the data directly. The handling of transformations, modeled as specialized attributes, is directly transferred to OpenGL; the engine interface has been extended by transformation methods in analogy to OpenGL. Therefore, the overall performance of a graphics application compared to a direct implementation based on the C API of OpenGL is not affected significantly: a small overhead results from 2 - 4 virtual function calls per shape necessary to invoke the painter.

10.2 Plug-Ins

The functionality of our reference implementation is extended by several plug-ins that incorporate graphics and geometry libraries into VRS. These plug-ins include the OpenGL-based tubing and extrusion library GLE [39], the graphics jungle implementation of the soft objects [37], engines for VRML, POV Ray, Radiance, and RenderMan, as well as graphical user interface bindings for Qt, Tcl/Tk, and Microsoft's MFC.

10.3 Applications

VRS is used in a variety of applications. It forms the visualization component for a computer animation system [7] that uses the OpenGL engine for real-time imaging and the RenderMan engine for high-quality output.

Currently, a real-time geo-visualization system is being developed on top of VRS [9]. In this application, digital terrain models are represented by specialized shape classes that provide level-of-detail modeling based on regular grids and TINs [2]. Specialized techniques implement multitexturing used to texture the terrain surface with multiple information layers [8] and a variety of visual effects typically used in flight simulation (including lens flares in the view plane and light reflection at the surfaces of the water bodies). Snapshots of this application are given in Figure 17.

Using the iterator concept it is easy to combine external terrain data and shape objects without duplicating the data. Due to the large amount of data, the application represents an excellent test case for the performance and practicability of our approach.

The built-in features of VRS simplified the implementation of the geo-visualization system. Especially useful was the possibility to include application-specific shapes, application-specific painters, and the design of application-specific rendering techniques. Without this possibility, the level-of-detail modeling for terrain geometry and the multitexturing of the terrain surface could not have been integrated as efficiently as in a native C implementation. As a consequence, many texture-related features already moved into the core of VRS since they proved to be generally useful.

Figure 17. Real-time terrain rendering based on VRS. Lens flares and hazy sun (left) are modeled as OpenGL-specific shapes. Multitexturing attributes are used to display multiple layers of terrain data such as a satellite texture and a road-network texture (right).

11 CONCLUSIONS

The generic rendering system represents a pragmatic approach for integrating and bundling the power of different rendering systems under a transparent, object-based rendering framework. It raises the level of abstraction of 3D graphics programming, enabling the quick and efficient development of 3D graphics applications using one or more rendering systems. Its functionality can be extended easily and efficiently by existing C and C++ graphics and geometry libraries due to its open architecture.

The object model of the generic rendering system makes no assumptions about the internal representation of graphics data, the internal structure of the rendering pipeline and the internal processing of rendering components. In particular, application data structures can be embedded into 3D graphics objects in an efficient way using iterators. The main functionality of shapes and attributes is decomposed into handlers, which allows the developer to configure and modify the functionality with respect to the visualization requirements of an individual application. The processing of rendering components is general enough to encapsulate multi-pass rendering techniques. The generic rendering system can be extended by the individual rendering features of an underlying rendering system. Especially most OpenGL features can be accessed without significant performance overhead. Therefore the generic rendering system can be used to implement time-critical graphics applications.

The demand for a generic rendering system is increasing because rendering systems have matured at an impressive speed in the past. Now, these systems represent an enormous variety of rendering techniques, but they are still difficult to use from a developer's perspective due to their complex, incompatible object models. The generic rendering system may help to overcome these obstacles and facilitate the integration of these systems. A reference implementation can be found at the web site of the Virtual Rendering System www.vrs3d.org.

ACKNOWLEDGEMENTS

The authors would like to thank Konstantin Baumann, Tobias Gloth, Oliver Kersting, and Florian Kirsch for their supportive work.

REFERENCES

[1] S. Amann, C. Streit, and H. Bieri, "BOOGA - A Component-Oriented Framework for Computer Graphics", *GraphiCon '97 Proceedings*, pp. 193-200, 1997.

[2] K. Baumann, J. Döllner, K. Hinrichs, and O. Kersting, "A Generic Data Structure for Real-Time Terrain Visualization", *Proc. IEEE Computer Graphics International '99*, pp. 85-92, 1999.

[3] E. Beier, "Issues on Hierarchical Graphical Scenes", *New Directions in Computer Graphics,* R. Veltkamp and E. Blake, eds., Springer-Verlag, pp. 3-12, 1995.

[4] E. Beier and U. Bozzetti, "A Generic Graphics Kernel and a Customized Derivative", *Proc. 6th EuroGraphics Workshop on Rendering*, 1995.

[5] P.R. Calder and M.A. Linton, "Glyphs: Flyweight Objects for User Interfaces", *Proc. ACM UIST*, pp. 92-101, 1990.

[6] S. Cunningham, N. Knolle Craighill, M.W. Fong, and J.R. Brown, *Computer Graphics Using Object-Oriented Programming*. Wiley Professional Computing, 1992.

[7] J. Döllner and K. Hinrichs, "Object-Oriented 3D Modeling, Animation and Interaction", *The Journal of Visualization and Computer Animation*, vol. 8, no. 1, pp. 33-64, 1997.

[8] J. Döllner, K. Baumann, and K. Hinrichs, "Texturing Techniques for Terrain Visualization", *Proc. IEEE Visualization*, pp. 227-234, 2000.

[9] J. Döllner and O. Kersting, "Dynamic 3D Maps as Visual Interfaces for Spatio-Temporal Data", *Proc. of the 8. ACM Symposium on Advances in Geographic Information Systems* (ACMGIS 2000), ACM Press, pp. 115-120, 2000.

[10] D.J. Duke and I. Herman, "Programming Paradigms in an Object-oriented Multimedia Standard", *Computer Graphics Forum*, vol. 17, no. 4, pp. 249-261, 1998.

[11] P.K. Egbert and W.J. Kubitz, "Application Graphics Modeling Support Through Object-Orientation", *IEEE Computer,* vol. 25, no. 10, pp. 84-91, 1992.

[12] P.K. Egbert, "Utilizing Renderer Efficiencies in an Object-Oriented Graphics System", *New Directions in Computer Graphics*, R. Veltkamp and E. Blake, eds., Springer-Verlag, pp. 13-22, 1995.

[13] C. Elliot, G. Schechter, R. Yeung, and S. Abi-Ezzi, "TBAG: A High Level Framework for Interactive, Animated 3D Graphics Applications", *Computer Graphics (Proc. SIGGRAPH `94)*, pp. 421-434, 1994.

[14] D.W. Fellner, "Extensible Image Synthesis", *Proc. 4th EuroGraphics Workshop on Object-Oriented Graphics*, pp. 1-18, 1994.

[15] L. Gritz and J.K. Hahn, "BMRT: A Global Illumination Implementation of the RenderMan Standard", *Journal of Graphics Tools*, vol. 1, no. 3, pp. 29-47, 1996.

[16] W. Heidrich and H.-P. Seidel, "Realistic, Hardware-accelerated Shading and Lighting", *Computer Graphics (Proc. SIGGRAPH '99)*, pp. 171-178, 1999.

[17] M.J. Kilgard, "A Practical and Robust Bump-Mapping Technique for Today's GPUs", *GDC 2000 - Advanced OpenGL Game Development*, 2000.

[18] M.J. Kilgard, "Improving Shadows and Reflections via the Stencil Buffer", *NVIDIA White Paper*, 2000.

[19] L. Koved and W.L. Wooten, "GROOP: An object-oriented toolkit for animated 3D graphics", *ACM SIGPLAN NOTICES OOPSLA '93*, vol. 28, no. 10, pp. 309-325, 1993.

[20] M.A. Linton, J.M. Vlissides, and P.R. Calder, "Composing User Interfaces with InterViews", *IEEE Computer*, vol. 22, no. 2, pp. 8-22, February 1989.

[21] T. McReynolds, D. Blythe, and B. Grantham, "Advanced Graphics Programming Techniques Using OpenGL", *SIGGRAPH 99 Course Notes*, 1999.

[22] Microsoft, Direct3D.

[23] S. Mohan, "The Fourth Generation of 3D Graphics APIs has arrived!", Sun Microsystems, *Java Markets White Paper*, 1998.

[24] D.R. Musser and A. Saini, *STL Tutorial and Reference Guide*. Addison-Wesley, 1996.

[25] M.S. Peercy, M. Olano, J. Airey, and J. Ungar, "Interactive Multi-Pass Programmable Shading", *Computer Graphics (Proc. SIGGRAPH 2000)*, pp. 425-432, 2000.

[26] POV Team, *Persistency of Vision Ray Tracer (POV-Ray).* Version 1.0, Technical Report, 1991.

[27] J. Rumbaugh, I. Jacobson, and G. Booch, *Unified Modeling Language Reference Manual.* Addison-Wesley, 1997.

[28] T. Saito and T. Takahashi, "Comprehensible Rendering of 3-D Shapes", *Computer Graphics (Proc. SIGGRAPH '92)*, vol. 24, no. 4, pp. 197-206.

[29] M. Shaw and D. Garlan, *Software Architecture. Perspectives on an Emerging Discipline.* Prentice Hall, 1996.

[30] Silicon Graphics Inc, *OpenGL Optimizer Programmer's Guide: An Open API for Large-Model Visualization*, 1998.

[31] P. Slusallek and H.-P. Seidel, "Object-Oriented Design for Image Synthesis", *Programming Paradigms in Computer Graphics*, Springer, pp. 23-34, 1995.

[32] P. Slusallek and H.-P. Seidel, "VISION – An Architecture for Global Illumination Calculations", *IEEE Transactions on Visualization and Computer Graphics*, vol. 1, no. 1, pp. 77-96, 1995.

[33] P. Strauss and R. Carey, "An Object-Oriented 3D Graphics Toolkit", *Computer Graphics (Proc. SIGGRAPH '92)*, vol 26, no. 2, pp. 341-349, 1992.

[34] H. Sowizral, "Scene Graphs in the New Millennium", *IEEE Computer Graphics and Applications*, vol. 20, no. 1, pp. 56-57, 2000.

[35] Sun Microsystems, *Java 3D API Specification.* Version 1.1, July 1998.

[36] C. Szyperski, *Component Software. Beyond Object-Oriented Programming.* Addison-Wesley, 1998.

[37] M. Tigges and B. Wyvill, "Texture mapping the blob-tree", *Proceedings of the Third Eurographics Workshop on Implicit Surfaces*, pp. 123-130, 1998.

[38] S. Upstill, *The RenderMan Companion. A Programmer's Guide to Realistic Computer Graphics.* Addison-Wesley, 1989.

[39] L. Vepstas, *The GLE Tubing and Extrusion Fact Sheet.* http://linas.org/gle

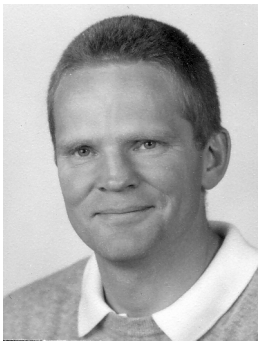
[40] G.J. Ward, "The RADIANCE Lighting Simulation and Rendering System", *Computer Graphics (Proc. of SIGGRAPH '94)*, pp. 459-472, 1994.

[41] T.F. Wiegand, "Interactive Rendering of CSG Models", *Computer Graphics forum,* vol. 15, no. 4, pp. 249-261, 1996.

[42] M. Woo, J. Neider, T. Davis, and D. Shreiner, *OpenGL Programming Guide*, 3rd ed. Addision-Wesley, 1999.

Jürgen Döllner received his diploma in mathematics 1993 from the University of Siegen, Germany, and his Ph. D. in 1997 from the University of Münster, Germany. He is a professor at the Hasso Plattner Institute at the University of Potsdam. His research interests include 3D rendering, information visualization, geo visualization, and 3D maps as well as software engineering and software architecture of computer graphics systems. He is a member of the IEEE.

Klaus Hinrichs received his diploma in mathematics with a minor in computer science in 1979 from the University of Hannover, Germany, and his Ph. D. in 1985 from the Swiss Federal Institute of Technology (ETH) in Zurich. He is a full professor of computer science at the University of Münster, Germany. His main research interests are in the areas of algorithms and data structures, especially for geometric computation, spatial data bases and visualization. He is a member of the ACM and the IEEE Computer Society.



Adresses

J. Döllner
Hasso-Plattner-Institute
University of Potsdam
Helmert-Str. 2-3, 14482 Potsdam, Germany
E-mail doellner@hpi.uni-potsdam.de
Phone ++49 331 5509 171
Fax ++49 331 5509 189

K. Hinrichs
Institute for Computer Science
University of Münster
Einsteinstr. 62, 48149 Münster, Germany
E-mail khh@uni-muenster.de
Phone ++49 251 8333752
Fax ++49 251 8333755

Appendix A

Diagram of Core Classes of the Virtual Rendering System (VRS)